Indian Institute of Technology Bombay

CS 769 Course Project

Combinatorial Optimization using Reinforcement Learning

May 6, 2022

Author	*Roll Number*
Tejas Pagare	190070067
Kaustubh Ponkshe	190100064

under the guidance of

Prof. Ganesh Ramakrishnan

Contents

1 Introduction

In this project, we explore the topic of combinatorial optimization. Combinatorial optimization is an optimization problem with discrete objects and the objective of the algorithm is to minimize or maximize a cost function. The problem is to find an optimal solution from a feasible set of discrete finite solutions. Some examples of combinatorial optimization include Travelling salesman-problem, minimum graph cut, set cover problem, etc. The applications of these problems are broad, ranging from logistics, kidney donation chaining to computer vision and water distribution. However the major problem in solving these algorithms is that they are NP hard problems. However, we can do better in the following ways :

1. Polynomial time exact solvers for some subsets of problems
2. Polynomial time algorithms which approximately solve the optimization upto some degree of performance worse than the optimal solution
3. Heuristic based algorithms which train using trial and error and learn heuristics

We will be focusing on the heuristic based algorithms and will look at a few examples of algorithms applied specifically to the travelling salesman problem. The development of excellent heuristics or approximation algorithms for NP-hard combinatorial optimization problems necessitates a great deal of specialised expertise and trial-and-error. Our aim is to automate these algorithms.

In many real-world applications, the same optimization issue is addressed over and over again on a regular basis, with the same problem structure but different data each time. This opens the door to developing heuristic algorithms that take use of the structure of such reoccurring situations.Heuristics are frequently quick, effective algorithms with no theoretical guarantees, and they may also need extensive problem-specific study and trial-and-error on the side of method inventors. The three described methods, however, rarely take use of a common feature of real-world optimization problems: instances of the same type of issue are addressed on a frequent basis, with the same combinatorial structure but mostly different input. In many cases, the values of the objective function's coefficients or restrictions may be regarded of as being sampled from the same underlying distribution.

In this work, we will compare various greedy and heuristic based strategies to solve the Travelling salesman problem. In particular, we will explore the methods involved and advantages of using deep learning to learn the huersitics and aid in particular algorithms.

Figure 1: Applications of Combinatorial Optimization in a nutshell [1]

2 Background

Suppose you are scientist and you are attending a conference. You want to visit all the posters and return back to where you started from by travelling the minimum distance. However, this we note is exactly the Travelling Salesman Problem, which is known to be NP-hard. To formalize this,

Travelling Salesman Problem:*"Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph exactly once"*

Consider a graph $G = (V, E)$ where $V = (v_1, v_2, ...v_n)$ is a set of n vertices and there is a cost function c_{ij} associated with each edge $e_{ij} \in E$. Now consider variables x_{ij} such that .

$$x_{ij} = \begin{cases} 1 \text{ if there is a path from i to j} \\ 0 \text{ otheriwse} \end{cases}$$

Then our optimization problem can be given as

$$\min \sum_{i=1}^{n} \sum_{i \neq j, j=1}^{n} x_{ij} c_{ij}$$

with the constraint that

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \text{ for every } j \text{ and } \sum_{i \neq j, j=1}^{n} x_{ij} = 1 \text{ for every } i$$

The problem can be represented as a graph, with each node representing the coordinate of the vertex. The graph is assumed to be fully connected. We will first go over the greedy algorithms used and their performance. Then we will list out some heuristic based algorithms where the heuristic shave been preset. Finally we will look at examples of learning heuristics for these algorithms.
Typically, many combinatorial problems can be represented by graphs and this makes it possible to share heuristics between related problems. In fact we will show that even the hyperparameters can be transferred across tasks. Took take few examples consider:

Minimum Vertex Cover (MVC):*" Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \iff u \in S$ or $v \in S$, and $|S|$ is minimized"*

Maximum Cut (MAXCUT):*"Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut-set $\sum_{(u,v) \in C} w(u, v)$ is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$."*

3 Greedy and Heuristic based algorithms

To start of for the task of solving the travelling salesman problem, we chose nearest neighbour algorithm (greedy algorithm) to see the results. Greedy algorithms are a class of algorithms which look for best in the short run, whether or not it is best in the long run.Greedy algorithms optimize locally, but not necessarily globally. They don't necessarily converge to optimal solution (although in some cases, we can give error bounds) but are usually very fast and simple. We will define the nearest neighbour algorithm as follows

"The Nearest-Neighbor Algorithm begins at any vertex and follows the edge of least weight from that vertex. At every subsequent vertex, it follows the edge of least weight that leads to a city not yet visited, until it returns to the starting point."

The average cost of such greedy algorithms has been found to be only a constant proportional worse than the optimal solution, however the worst case solutions can be abitrarily bad.

Besides this, we will also go over two heuristic based (but not learning based) algorithms, namely the nearest insertion and the farthest insertion algorithms.

Algorithm 1 Nearest Insertion Algorithm

```
**Select** $v_1 \in V$, Intitialize subgraph $S = \phi$
Find vertex $v_j = \underset{v_j}{\text{argmin}}\ c_{1j}$        ▷ vertex closest to $v_1$
$S = \{v_1, v_j, v_1\}$        ▷ partial solution
**for** *t=1 to n-2* **do**
    Find vertex $v_k = \underset{v_i \in S}{\text{argmin}}\ c_{ik}$        ▷ vertex closest to all nodes in $S$
    Find arc $(i, j) = \underset{v_i, v_j \in S}{\text{argmin}}\ c_{ik} + c_{kj} - c_{ij}$
    Update $S = \{v_1, \ldots, v_j, v_k, v_i, \ldots, v_1\}$        ▷ insert $v_k$ between $v_i$ and $v_j$
**end**
```

The farthest insertion can be given by chosing the farthest vertex rather than the nearest one.

3.1 Experimental Results

Following are experimental results for the Traveling Salesman Problem

Nodes	Nearest Neighbour	Nearest Insertion	Farthest Insertion
20	5.106	5.248	5.246
50	7.293	7.378	8.076

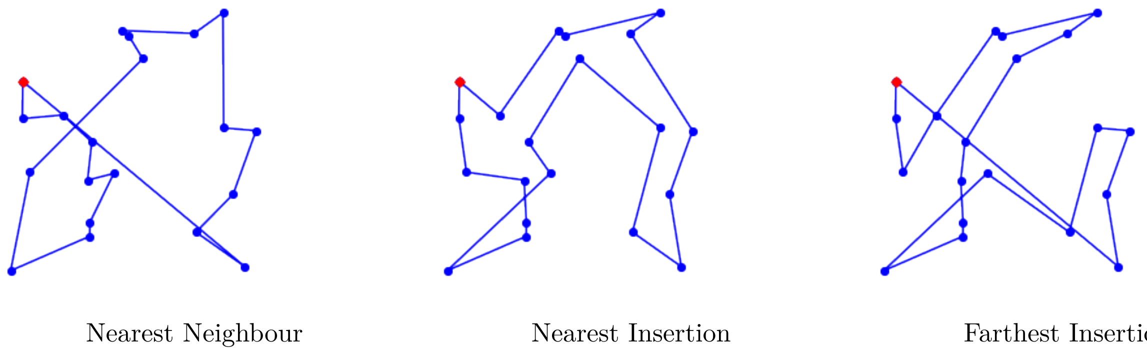


Figure 2: 20 nodes
red vertex indicates the starting and hence ending point

4 Learning Methods

As can be seen from the above algorithms it is quite cumbersome to come with a good heuristic to solve CO problems and in most cases it is problem dependent. Now, we will move to some techniques which aims to learn the heuristic for CO problems. This is where we encapsulate Reinforcement Learning where a particular CO problem can be adopted as an environment which provides rewards/cost for each action taken by an agent and the agent has to explore and exploit the information gathered from the environment interaction in order to optimize the objective.

4.1 Reinforcement Learning

First of all, to use Reinforcement Learning, a CO problem has to modeled as an sequential decision problem in particular as a Markov Decision Process exploiting the Markov Property which holds in most scenarios

Definition 4.1 (Controlled Markov Property)**.**

$$\Pr\{X_n = j | X_m, U_m, m < n\} = \Pr\{X_n = j | X_{n-1} = i, U_{n-1} = a\}$$

which says that state X_n is independent of past states and actions given the last state X_{n-1} and action U_{n-1}.

Definition 4.2 (Markov Decision Process)**.** A MDP consists of

- A set of states $\mathcal{S}$, set of actions $\mathcal{A}$ for moving from a state to another. $\mathcal{S}$ and $\mathcal{A}$ can be both finite or infinite.
- Transition Probabilities P which is defined the probability distribution over next states given the current state and current action where $P_{ij}(a) = \Pr\{X_{n+1} = j | X_n = i, U_n = a\}$.
- A reward function which is defined as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\mathcal{R}_s^a$ or $r(s, a)$ is the expected reward of taking action a in state s.

Therefore a MDP is simply given as a pair $(\mathcal{S}, \mathcal{A}, \mathcal{R})$.

A policy $\pi : \mathcal{S} \to \mathcal{A}$ is a distribution over actions given states $\pi(a|s) = \Pr\{U_n = a | X_n = s\}$.
For stationary policy we have $U_n \sim \Pr(\cdot | X_n = s)\ \forall t > 0$.
Any policy induces a Markov chain whose transition probability or stochastic matrix is denoted as $P(\pi) \in [0,1]^{|\mathcal{S}| \times |\mathcal{S}|}$ where each entry $P_{ij}(\pi) = \sum_{a \in \mathcal{A}} \pi(a|s) p(j|i, a)$. This Markov chain is denoted as $(\mathcal{S}, \mathcal{P}(\pi))$.

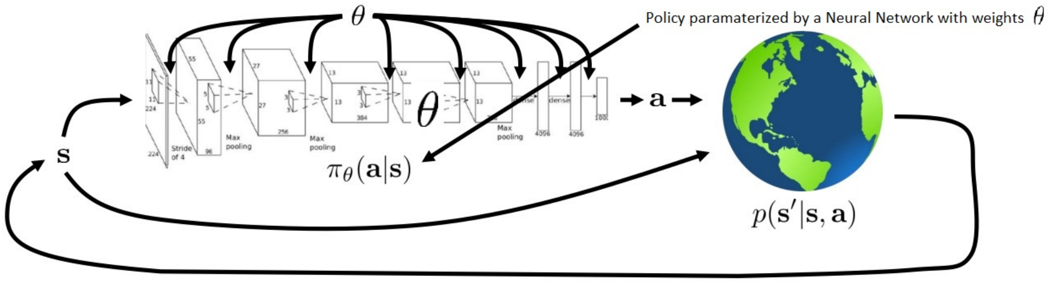


Figure 3: Agent-Environment Interaction [2]

Objective of Reinforcement Learning is to maximize (minimize) the expected long run reward (cost) formally defined as finding

$$\theta^\star = \underset{\theta}{\operatorname{argmax}} \mathbb{E}_{\tau \sim p_\theta(\tau)} \Big[\sum_t r(s_t, a_t) \Big]$$

where

$$p_\theta(s_1, a_t, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t)$$

where T can be infinite.

Now, we will explore a particular Reinforcement Learning algorithm called REINFORCE [3] which belongs to the family of Policy Gradient algorithms

4.1.1 REINFORCE Algorithm

We will first state a generalization of policy gradient theorem with a baseline (reduces variance)

Theorem 4.1 (Policy Gradient Theorem).

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \Big(Q_\pi(s,a) - b(s)\Big)\nabla \pi_\theta(a|s) = \mathbb{E}_\pi\Big[\sum_a \Big(Q_\pi(s,a) - b(s)\Big)\nabla \pi_\theta(a|s)\Big] \tag{1}$$

where $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\Big[\sum_t r(s_t, a_t)\Big]$ and

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\Big]$$

is the Q-value function for policy π and $\mu(s)$ is a state distribution satisfying $\mu(s) \geq 0 \; \forall \; s$ and $\sum_s \mu(s) = 1$.

In Eq.(1), the baseline $b(s)$ can be in general any random variable depending on s and the most common and natural choice is to use an estimate of the state-value function i.e. $\hat{V}(s)$ generally a parameterized function $\hat{V}(s, \mathbf{w})$ with weight vector $\mathbf{w} \in \mathbb{R}^d$.

The policy-gradient methods seek to maximize the perform ace using the stochastic gradient *ascent* in J as follows:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$$

where $\widehat{\nabla J(\theta_t)}$ is the stochastic estimate of the actual gradient of J w.r.t θ

REINFORCE is a policy gradient method which is based Monte-Carlo estimates to update the policy parameters θ, the update is as follows

$$\theta_{t+1} = \theta_t + \alpha\Big(G_t - b(S_t)\Big)\frac{\nabla \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}$$

Algorithm 2 REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi_\theta(a|s)$, a differentiable state-value function parameterization $\hat{V}(s,w)$
Algorithm parameters: step sizes $\alpha_1 > 0, \alpha_2 > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value parameters $w \in \mathbb{R}^d$
for each episode do
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi_\theta(\cdot|\cdot)$
    for step t=0,1,...,T-1 do
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \hat{V}(s,w)$
        $w \leftarrow w + \alpha_1 \delta \nabla \hat{V}(S_t, w)$
        $\theta \leftarrow \theta + \alpha_2 \gamma^t \delta \nabla \ln \pi_\theta(A_t|S_t)$
    end
end
```

4.2 Attention Based REINFORCE Algorithm

Now, we will explore the paper [4] which uses an encoder-decoder based architecture for the policy and applies REINFORCE for optimization.

Consider for the n-node graph problem instance s, the solution tour $\pi = (\pi_1, \ldots, \pi_n)$ as the permutation of nodes, $\pi_t \in \{1, \ldots, n\}$. The aim is to find a stochastic policy factorized using $p_\theta(\pi|s) = \prod_{t=1}^{n} p_\theta(\pi_t|s, \pi_{1:t-1})$.

The encoder produces the embedding for each node and the decoder produces the sequence π of input nodes one at a time using the encoder node embeddings.

4.2.1 Attention Model

Here, we consider the an encoder-decoder architecture similar to Transformer [5].

Encoder:

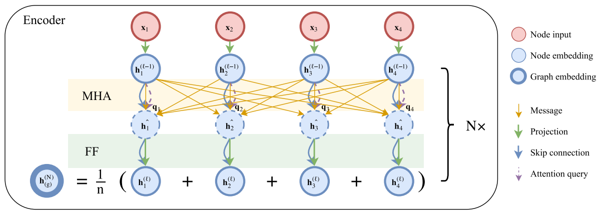


Figure 4: Attention based encoder

Here for encoder we have
x_i is d_x dimensional input feature for node i e.g. x,y coordinates for TSP.
$\mathbf{h}_i^{(0)}$ is learned d_h dimensional node embedding with parameters W^x and b^x as $\mathbf{h}_i^{(0)} = W^x x_i + b^x$
Output is node embedding for each node i, $\mathbf{h}_i^{(N)}$ and graph embedding $\bar{\mathbf{h}}^{(N)} = \dfrac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i^{(N)}$
$\mathbf{h}_i^{(l)}$ is the node embedding produced by the layer $l \in \{1, \ldots, N\}$
Each attention layer contains a Multi-Head Attention (MHA) and fully connected feed-forward (FF) layer same as the Transformer architecture and skip-connection and batch normalization (BN):

$$\hat{\mathbf{h}}_i = \mathrm{BN}^{\ell}\left(\mathbf{h}_i^{(\ell-1)} + \mathrm{MHA}_i^{\ell}\left(\mathbf{h}_1^{(\ell-1)}, \ldots, \mathbf{h}_n^{(\ell-1)}\right)\right) \tag{2}$$

$$\mathbf{h}_i^{(\ell)} = \mathrm{BN}^{\ell}\left(\hat{\mathbf{h}}_i + \mathrm{FF}^{\ell}(\hat{\mathbf{h}}_i)\right). \tag{3}$$

Decoder:
Decoder outputs π_t sequentially based on embeddings from the network and $\pi_{t'}$ for $t' < t$.

The graph is augmented with a special context node (c) which essentially tries to represent the decoding context. The context embedding is given as

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \mathbf{h}_{\pi_1}^{(N)}\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{v}^{\mathrm{l}}, \mathbf{v}^{\mathrm{f}}\right] & t = 1. \end{cases} \tag{4}$$

where $[\cdot, \cdot, \cdot]$ is the horizontal concatenation operator and $\mathbf{v}^{\mathrm{l}}, \mathbf{v}^{\mathrm{f}}$ are learned.

Now finally we perform two attention steps, first one uses Multi-Head Attention between context node and other nodes and later one is a single head attention to calculate the log-probabilities of the tour sequence.

To compute the context node encoding $\mathbf{h}_{(c)}^{(N+1)}$ the keys and values are computed for each node using the node embeddings $\mathbf{h}_i^{(N)}$, and a single query $\mathbf{q}_{(c)}$ (per head) is computed:

$$\mathbf{q}_{(c)} = W^Q \mathbf{h}_{(c)} \quad \mathbf{k}_i = W^K \mathbf{h}_i, \quad \mathbf{v}_i = W^V \mathbf{h}_i. \tag{5}$$

To calculate the log-probabilities we first compute compatibilities $u_{(c)j}$, after clipping the result (before masking!) within $[-C, C]$ (C = 10) using tanh:

$$u_{(c)j} = \begin{cases} C \cdot \tanh\left(\dfrac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_{\mathrm{k}}}}\right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{6}$$

We interpret these compatibilities as unnormalized log-probabilities (logits) and compute the final output probability vector **p** using a softmax:

$$p_i = p_\theta(\pi_t = i|s, \pi_{1:t-1}) = \frac{e^{u_{(c)i}}}{\sum_j e^{u_{(c)j}}}. \quad (7)$$

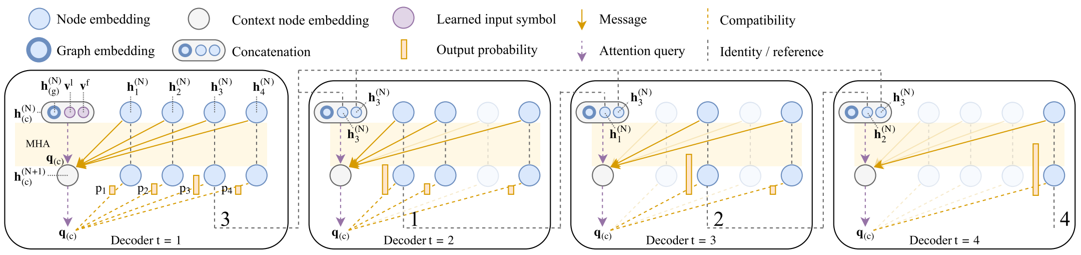


Figure 5: Decoder with output tour as π =(3,1,2,4)

4.2.2 REINFORCE

For a given problem instance s, the encoder-decoder architecture outputs $_\theta(\pi|s)$ from which we sample a tour π and define the loss as we sample a policy π (permutation of nodes) to computed the loss defined as

$$\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[L(\pi)]$$

We optimize $\mathcal{L}$ using gradient-ascent most commonly known as REINFORCE where we have the gradient of the loss as

$$\nabla\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[(L(\pi) - b(s))\nabla \log p_\theta(\pi|s)]$$

A good baseline $b(s)$ reduces the gradient variance!
We use 2 baselines for the experimentation:

1. greedy rollout baseline: defined as the cost of a solution of a deterministic greedy rollout policy by the best model so far
2. critic: a function $\hat{V}(s, w)$ of state s, parameterized by w, which is learned using gradient ascent iteration similar to original REINFORCE iteration

Intuition behind greedy rollout baseline: The policy gradient in REINFORCE has the term $L(\pi) - b(s)$ which is negative if the sampled solution π is better that the greedy sampled baseline causing actions to be reinforced (hence the name) and vice versa. This indicated the improvement of the model from its *greedy self*!

The motivation behind choosing a baseline that it should estimate the difficulty of the problem instance s.
Based on the observation that the difficulty of a problem instance depends on the performance of an algorithm used for training motivates that the baseline should be defined by the model during training.
To eliminate the variance, the baseline is chosen greedily to the action with maximum probability.

Now, it is important to note that during the training process, model keeps changing and hence the baseline and hence it order to stabilize the basline it is kept frozen for fixed number of steps every epoch.
The parameters associated with baseline policy θ^{BL} is changed to policy parameters θ at the end of every epoch if current training policy is better compared to the baseline policy according to a paired t-test with significance α.

The REINFORCE algorithm is fully described using the following pseudo-code

Algorithm 3 REINFORCE with Rollout Baseline

```
Input: number of epochs $E$, steps per epoch $T$, batch size $B$, significance $\alpha$
Init $\theta$, $\theta^{\mathrm{BL}} \leftarrow \theta$
for $epoch = 1, \ldots, E$ do
    for $step = 1, \ldots, T$ do
        $s_i \leftarrow \mathrm{RandomInstance}()\ \forall i \in \{1, \ldots, B\}$
        $\pi_i \leftarrow \mathrm{SampleRollout}(s_i, p_\theta)\ \forall i \in \{1, \ldots, B\}$
        $\pi_i^{\mathrm{BL}} \leftarrow \mathrm{GreedyRollout}(s_i, p_{\theta^{\mathrm{BL}}})\ \forall i \in \{1, \ldots, B\}$
        $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left(L(\pi_i) - L(\pi_i^{\mathrm{BL}})\right) \nabla_\theta \log p_\theta(\pi_i)$
        $\theta \leftarrow \mathrm{Adam}(\theta, \nabla \mathcal{L})$
    end
    if $OneSidedPairedT\text{-}Test(p_\theta, p_{\theta^{BL}}) < \alpha$ then
        $\theta^{BL} \leftarrow \theta$
    end
end
```

5 Experimental Results

We carry several experiments for ablation study of the Attention based REINFORCE algorithm. The experiments are carried out for Traveling Salesman Problem with 20 nodes.

The algorithm takes about 5 minutes to learn for 20 nodes on a single NVIDIA 1660Ti GPU.

1. Using different baselines:

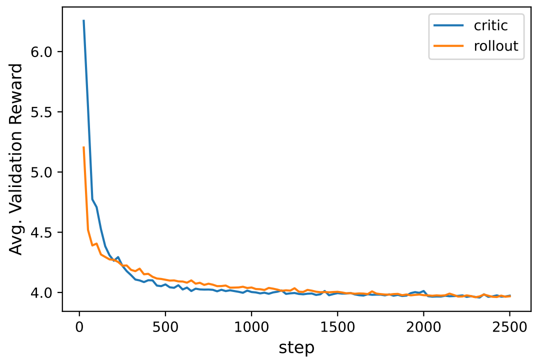
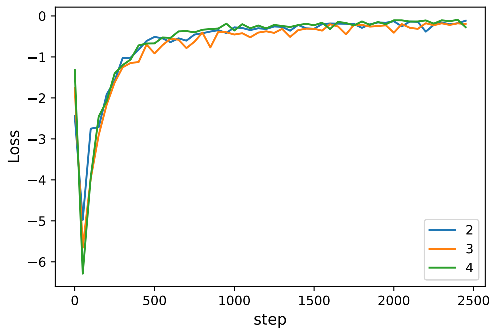


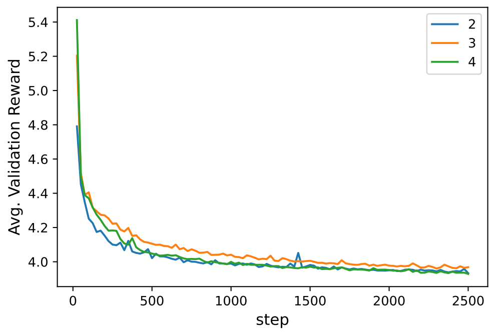
Figure 6: Different baseline: greedy rollout and critic

   As can be seen from the figure, the average tour length tend to lower in case of the greedy rollout baseline as compared to the learned critic baseline which validates our intuition.

2. The next study involves the effect of different number of layers in the encoder part, which indicate a trade-off between quality of the results and computational complexity (runtime) of the model and the best is achieved with $N = 3$.



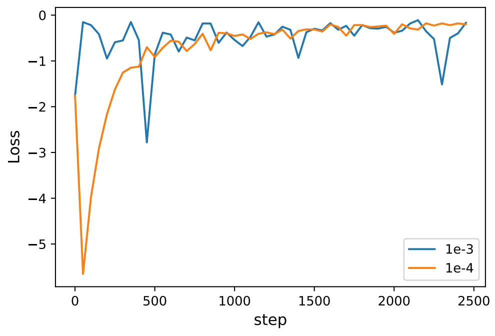
(a) Learning Loss



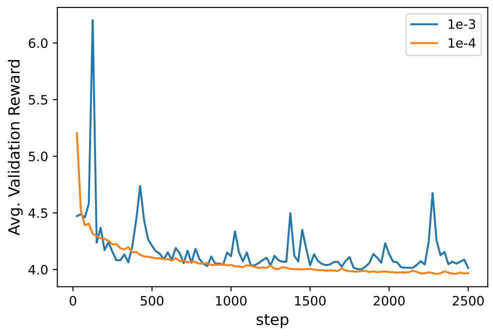
(b) Average Reward

Figure 7: Different Encoder Layers

3. Lastly, we validate that with higher learning rate of 0.001 the learning becomes unstable and the optimal is found to be 0.0001.



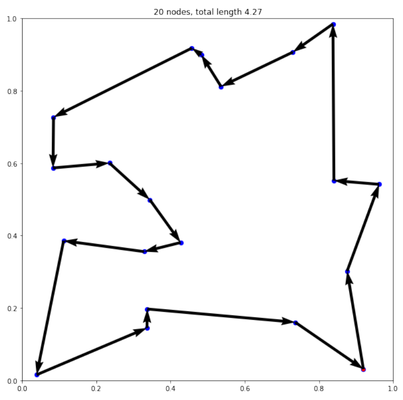
(a) Learning Loss



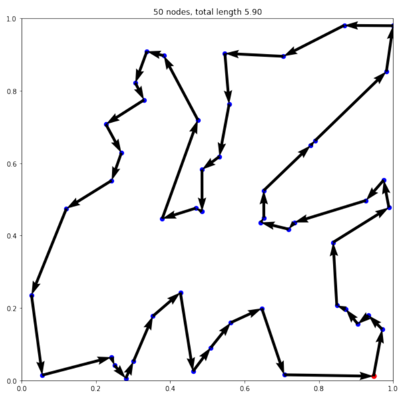
(b) Average Reward

Figure 8: Different Learning Rate

Finally we show surprising results for the TSP problem with 20 and 50 nodes



(a) 20 Nodes
Tour length 4.27



(b) 50 Nodes
Tour length 5.90

Figure 9: Solving Traveling Salesman Problem

Our codebase for the experimentation can be found here. We have used the author's codebase for ablation study of the Attention based REINFORCE algorithm.

References

[1] Recent advances in integrating Machine learning & Combinatorial optimization AAAI-21 Tutorial. `https://sites.google.com/view/ml-co-aaai-21/`.

[2] Lecture 5: Policy Gradients, Deep RL course by Sergey Levine. `http://rail.eecs.berkeley.edu/deeprlcourse-fa19/static/slides/lec-5.pdf`.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[4] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2018.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.